We're doing it live: A multi-method empirical study on continuous experimentation

Schermann, Gerald ; Cito, Jürgen ; Leitner, Philipp ; Zdun, Uwe ; Gall, Harald C

Abstract: Context Continuous experimentation guides development activities based on data collected on a subset of online users on a new experimental version of the software. It includes practices such as canary releases, gradual rollouts, dark launches, or A/B testing. Objective Unfortunately, our knowledge of continuous experimentation is currently primarily based on well-known and outspoken industrial leaders. To assess the actual state of practice in continuous experimentation, we conducted a mixed-method empirical study. Method In our empirical study consisting of four steps, we interviewed 31 developers or release engineers, and performed a survey that attracted 187 complete responses. We analyzed the resulting data using statistical analysis and open coding. Results Our results lead to several conclusions: (1) from a software architecture perspective, continuous experimentation is especially enabled by architectures that foster independently deployable services, such as microservices-based architectures; (2) from a developer perspective, experiments require extensive monitoring and analytics to discover runtime problems, consequently leading to developer on call policies and influencing the role and skill sets required by developers; and (3) from a process perspective, many organizations conduct experiments based on intuition rather than clear guidelines and robust statistics. Conclusion Our findings show that more principled and structured approaches for release decision making are needed, striving for highly automated, systematic, and data- and hypothesis-driven deployment and experimentation.

DOI: https://doi.org/10.1016/j.infsof.2018.02.010

We're Doing It Live: A Multi-Method Empirical Study on Continuous Experimentation

Gerald Schermann[a,*], Jürgen Cito[a], Philipp Leitner[a],
Uwe Zdun[b], Harald C. Gall[a]

[a] *Department of Informatics, University of Zurich, Switzerland*
{schermann, cito, leitner, gall}@ifi.uzh.ch
[b] *University of Vienna, Austria*
uwe.zdun@univie.ac.at

Abstract

Context: Continuous experimentation guides development activities based on data collected on a subset of online users on a new experimental version of the software. It includes practices such as canary releases, gradual rollouts, dark launches, or A/B testing.
Objective: Unfortunately, our knowledge of continuous experimentation is currently primarily based on well-known and outspoken industrial leaders. To assess the actual state of practice in continuous experimentation, we conducted a mixed-method empirical study.
Method: In our empirical study consisting of four steps, we interviewed 31 developers or release engineers, and performed a survey that attracted 187 complete responses. We analyzed the resulting data using statistical analysis and open coding.
Results: Our results lead to several conclusions: (1) from a software architecture perspective, continuous experimentation is especially enabled by architectures that foster independently deployable services, such as microservices-based architectures; (2) from a developer perspective, experiments require extensive monitoring and analytics to discover runtime problems, consequently leading to developer on call policies and influencing the role and skill sets required by developers; and (3) from a process perspective, many organizations conduct experiments based on intuition rather than clear guidelines and robust statistics.
Conclusion: Our findings show that more principled and structured approaches for release decision making are needed, striving for highly automated, systematic, and data- and hypothesis-driven deployment and experimentation.

Keywords: `release engineering, continuous deployment, continuous experimentation, empirical study`

1. Introduction

Many software developing organizations are looking into ways to further speed up their release processes and to get their products to their customers faster [1]. One instance of this is the current industry trend to "move fast and break things", as made famous by Facebook [2] and in the meantime adopted by a number of other industry leaders [3]. Another example is continuous delivery and deployment (CD) [4]. *Continuous delivery* is a software development practice where software is built in such a way that it can be released to production at any time, supported by a high degree of automation [5]. *Continuous deployment* goes one step further; software is released to production as soon as it is ready, i.e., passing all quality gates along the deployment pipeline. These practices pave the way for controlled continuous experimentation (e.g., A/B testing [6], canary releases [4]), which are a means to guide development activities based on data collected on a subset of online users on a new *experimental* version of the software. Unfortunately, our knowledge of continuous experimentation practices is currently primarily based on well-known and outspoken industrial leaders [6, 7]. This is a cause for concern for two reasons. Firstly, it raises the question to what extent our view of these practices is coined by the peculiarities and needs of a few innovation leaders, such as Microsoft, Facebook, or Google. Secondly, it is difficult to establish what the broader open research issues in the field are.

Hence, we conducted a mixed-method empirical study, in which we interviewed 31 software developers and release engineers from 27 companies. To get the perspective of a broader set of organizations, we specifically focused on a mix of different team and company sizes and domains. However, as continuous experimentation is especially amenable for Web-based applications, we primarily selected developers or release engineers from companies developing Web-based applications for our interviews. We

*Corresponding author

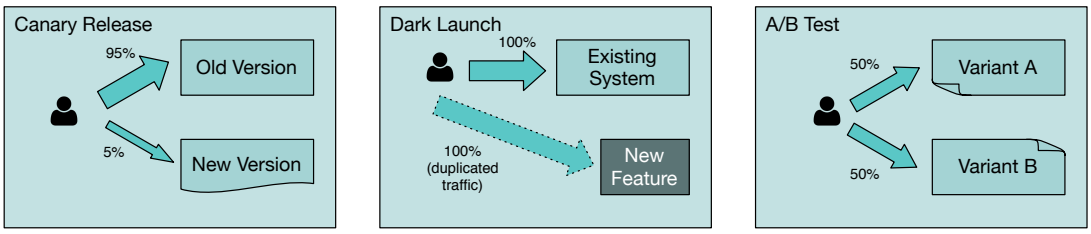

Figure 1: Overview of canary releases, dark launches, and A/B testing.

combined the gathered qualitative interview data with an online survey, which attracted a total of 187 complete responses. The design of the study was guided by the following research questions.

***RQ1**: What principles and practices enable and hinder organizations to leverage continuous experimentation?*

We identified the preconditions for setting up and conducting continuous experiments. Continuous experimentation is facilitated through a high degree of deployment automation and the adoption of an architecture that enables independently deployable services (e.g., microservices-based architectures [8]). Important implementation techniques include *feature toggles* [9] and *runtime traffic routing* [10]. Experimenting on live systems requires more insight into operational characteristics of these systems. This requires extensive monitoring and safety mechanisms at runtime. Developer on call policies are used as risk mitigation practices in an experimentation context. Experiment data collection and interpretation is essential. However, not all teams are staffed with experts in all relevant fields, we have seen that these teams can request support from internal consulting teams (e.g., data scientists, DevOps engineers, or performance engineers).

***RQ2**: What are the different flavors of continuous experimentation and how do they differ?*

Having insights into the enablers and hindrances of experimentation, we then investigated how companies make use of experimentation. Organizations use different flavors of continuous experimentation for different reasons. *Business-driven experiments* are used to evaluate new functionality from a business perspective, first and foremost using A/B testing [6]. *Regression-driven experiments* are used to evaluate non-functional aspects of a change in a production environment, i.e., validate that a change does not introduce an end user perceivable regression. In our study, we have observed differences in these two flavors concerning their main goals, evaluation metrics, how their data is interpreted, and who bears the responsibility for different experiments. We have also seen commonalities in how experiments are technically implemented and what their main obstacles of adoption are.

Based on the outcomes of our study, we propose a number of promising directions for future research. Given the importance of architecture for experimentation, we argue that further research is required on architectural styles that enable continuous experimentation. Further, we conclude that practitioners are in need of more principled approaches to release decision making (e.g., which features to conduct experiments on, or which metrics to evaluate).

The rest of this paper is structured as follows. In Section 2, we introduce common continuous experimentation practices. Related previous work is covered in Section 3. Section 4 gives more detail on our chosen research methodology, as well as on the demographics of our study participants and survey respondents. The main results of our research are summarized in Sections 5 and 6, while more details on the main implications and derived future research directions are given in Section 7. Finally, we conclude the paper in Section 8.

2. Background

Adopting CD, thus increasing release velocity, has been claimed to allow companies to take advantage of early customer feedback and faster time-to-market [1]. However, moving fast increases the risk of rolling out defective versions. While sophisticated test suits are often successful in catching functional problems in internal test environments, performance regressions are more likely to remain undetected, hitting surface only under production workloads [11]. Techniques such as user acceptance testing help companies estimate how users appreciate new functionality. However, the scope of those tests is limited and allows no reasoning about the demand of larger populations. To mitigate these risks, companies have started to adopt various continuous experimentation practices, most importantly canary releases, gradual rollouts, dark launches, and A/B testing. We provide a brief overview of these experimentation practices in Section 2.1, followed by an introduction to two common techniques how these practices can be implemented in Section 2.2.

2.1. Experimentation Practices

Figure 1 illustrates the practices of canary releases, dark launches, and A/B testing.

Canary Releases. Canary releases [4] are a practice of releasing a new version or feature to a subset of customers only (e.g., randomly selecting 5% of all customers in a geographic region), while the remaining customers

continue using the stable, previous version of the application. This type of testing new functionality in production limits the scope of problems if things go wrong with the new version.

Dark Launches. Dark, or shadow, launching [2, 12] is a practice to mitigate performance or reliability issues of new or redesigned functionality when facing production-scale traffic. New functionality is deployed to production environments without being enabled or visible for any users. However, in the backend, "silent" queries generated based on production traffic are forwarded to the "shadow" version. This provides insights into how the feature would be behaving in production, without actually impacting users.

Gradual Rollouts. Gradual rollouts [4] are often combined with other continuous experimentation practices, such as canary releases or dark launches. The number of users assigned to the newest version is gradually increased (e.g., increase traffic routed to the new version in 5% steps) until the previous version is completely replaced or a predefined threshold is reached.

A/B Testing. A/B testing [6] comprises running two or more variants of an application in parallel, which only differ in an isolated implementation detail. The goal is to statistically evaluate, usually based on business metrics (e.g., conversion rate), which of those versions performed better, or whether there was a statistically significant difference at all.

2.2. Implementation Techniques

The two common implementation techniques for conducting experiments are feature toggles and runtime traffic routing.

Feature Toggles. Feature toggles [9] are a code-level experimentation technique. In their simplest form, they are conditional statements in the source code deciding about which code block to execute next (e.g., whether a certain feature is enabled for a specific user or user group).

```
1 if isEnabled('newFeature', $user)
2 # code block containing new feature
3 else
4 # code block containing old functionality
5 end
```

Runtime Traffic Routing. Runtime traffic routing is a network-level experimentation technique. Multiple versions of an application or service run in parallel (e.g., as virtual machines, cloud instances, or containers). Depending on filter criteria applied on user requests (e.g., header information such as cookies, device identifiers), dynamically configured (network-level) components (e.g., proxies) decide to which concrete version of an application or service requests should be forwarded. A special type of traffic routing are blue/green deployments [13], which include two or more active versions at the same time, but only one serves production traffic.

3. Related Work

Release engineering and CD is currently a popular topic of study in software engineering and data science. We categorized related work into (1) research related to continuous integration (CI) as a prerequisite for CD and continuous experimentation, (2) research related to CD including its adoption and challenges involved, and (3) research covering continuous experimentation practices and experience reports.

3.1. Continuous Integration

Continuous Integration (CI) as prerequisite for CD has been studied extensively in recent years. Vasilescu et al. [14] studied the effects of CI in the context of open source projects that use pull requests on GitHub. Hilton et al. [15] conducted a detailed analysis of the usage of CI in open source projects and showed that CI supports more frequent releases and is widely adopted by popular software projects. Recently, Hilton et al. [16] reported on an empirical study investigating the barriers and needs developers face when using CI including trade-offs related to security, flexibility, and assurance. Similarly, Debbiche et al. [17] reported on the challenges a telecommunications company faced on their way to adopt CI. Brandtner et al. [18] have found that integrating build information from multiple sources across the CI tool chain can support developers to stay aware about the quality and health state of a software system. Ståhl and Bosch [19] proposed a model for documenting the practice of CI derived from a systematic literature review and illustrated its application on an industry case study. In the scope of CI, there are also a multitude of research on software builds and testing. Beller et al. [20] studied how central testing is to the CI process and analyzed more than 2 million builds on the Travis CI service. Similarly, Rausch et al. [21] investigated the factors that lead to build failures on Travis CI. A similar research question was also investigated by Vassallo et al. [22], who additionally compared build failures of OSS projects with projects of a financial organization, leading to a taxonomy of build failures.

3.2. Continuous Delivery and Deployment

Roadmaps and Literature Reviews. Adams and McIntosh [23] provided a roadmap for future research on CD and release engineering practices. Similarly, Rodriguez et al. [24] conducted a systematic literature review on CD research articles and addressed potential fields for future research. In their systematic literature review, Shahin et al. [25] classified available approaches and tools in the context of CI and CD. Moreover, they identified challenges, practices, and gaps for future research considering the current state of CI and CD. Rahman et al. [26] conducted a qualitative analysis of CD practices performed by 19 software companies by analyzing company blogs and similar online texts. However, they did not conduct interviews or a formal survey beyond what is already available in

blogs. In their white paper, Forrester [27] conducted a survey with 325 business and IT executives and showed that many companies have a low level of maturity when it comes to CD, and consequently are not able to keep innovation as high as business aims for.

DevOps. There are also studies on the state of the art in DevOps. The most authoritative source on this comes from Puppet Lab [28], a provider of Infrastructure-as-Code tooling, which releases annual reports on the state of DevOps. Academic studies in this field include our own previous work [29] about integrating runtime monitoring data from production environments into developer tools, but also the work conducted in the CloudWave project [30]. Lwakatare et al. [31] combined a literature survey and practitioner interviews to investigate the DevOps "phenomenon". They identified collaboration, automation, measurement, and monitoring as the characterizing DevOps elements. In a more recent study, Lwakatare et al. [32] studied the relationship of DevOps to agile, lean, and CD approaches. Shahin et al. [33] identified different types of team structures by investigating how development and operations teams are organized in the industry for adopting CD practices.

CD Adoption and Challenges. Recent research has comprised multiple studies on the challenges companies face when adopting CD. Leppanen et al. [34] and Olsson et al. [35] conducted studies with multiple companies discussing technical and organizational challenges, and their state of CD adoption. Similarly, Chen [1], and Neely and Stolt [36] provide experience reports from a perspective of a single case study company, the obstacles they needed to overcome and the benefits they gained by establishing CD-based release processes. Claps et al. [37] identified social challenges that companies face, and present mitigation strategies. Recently, Chen [38] presented six strategies to overcome adoption challenges and in addition proposed possible directions for future research. Bellomo et al. [39] investigated architectural decisions companies take to enable CD and introduced deployability and design tactics. Itkonen et al. [40] investigated the adoption of CD in a single case study company and report on the benefits it enables for both customers and developers. Fitzgerald and Stol [41] reported on the need for a tighter collaboration between software development and business strategy to enable continuous planning. In previous work [42], we derived a model based on the trade-off between release confidence (i.e., the effort companies put into quality gates throughout the development process) and the velocity of releases (i.e., the pace with which they can release new versions).

As Facebook is one of the main drivers in the professional developer scene surrounding CD and continuous experimentation, the company is also commonly the subject of related studies. Feitelson et al. [2] describe practices Facebook adopted to release on a daily basis. In a recent work, Savor et al. [43] compared CD experiences at Facebook and OANDA and revealed that CD allows scaling in both the number of developers and code base sizes without decreasing productivity.

3.3. Continuous Experimentation

Experience Reports. There are also a multitude of academic publications discussing how key industrial players conduct continuous experiments. Tang et al. [12] give insights how Facebook manages multiple versions running in parallel (e.g., using A/B testing) with a sophisticated configuration-as-code approach. There are also experience reports of Microsoft [6] and Google [7] on how they conduct experiments at a large scale. These works frame this research as a data science rather than a software or release engineering topic. In contrast, Kevic et al. [44] investigated experimentation at Microsoft from a software engineering perspective. Using Bing as a case study, they investigated the complexity of the experimentation process and results show that code changes for experiments are four times larger than other code changes. Similarly, Fabijan et al. [45] investigated the evolution of experimentation at Microsoft and presented a model detailing technical, organizational, and business evolution to provide a guidance towards data-driven experimentation.

Process and Design. Fagerholm et al. [46] investigated the preconditions for setting up an experimentation system and characterized software instrumentation to collect, analyse, and store data as one of the challenges for experimentation. Bakshy and Frachtenberg [47] provide guidelines for correctly designing and analyzing benchmark experiments. Bakshy et al. [48] proposed a language for describing online field experiments, including A/B testing, at Facebook. Kohavi et al. [49] provided a practical guide for conducting experiments. Tarvo et al. [50] built a tool for automated canary testing incorporating the automated collection and analysis of metrics using statistics. Tamburrelli and Margara [51] rephrase A/B testing as a search-based software engineering problem targeting automation by relying on aspect-oriented programming and genetic algorithms.

Implementation Techniques and Tooling. Rahman et al. [52] analyzed the usage and evolution of feature toggles in 39 releases of Google Chrome and discussed their strengths and drawbacks. Recently, Veeraraghavan et al. [10] described how Facebook uses a tool called *Kraken* to control (i.e. route) live user traffic on various levels (i.e., data center, server) to identify and resolve bottlenecks across their application ecosystem. Our own tooling, *Bifrost* [53], supports the specification of experiments in a domain-specific language and uses runtime traffic routing for redirecting user requests to the right service versions.

3.4. Open Issues

Despite this significant body of work, we observe some relevant gaps. Primarily, the existing body of research uses case study research based on one, or very few, companies.

In our work, we conduct a mixed-method study based on a larger sample size. Further, we focus on the software developer's or release engineer's point of view, rather than the perspective of managers, product owners, or data scientists. Lindgren and Münch [54] recently did a step into a similar direction, focusing on a manager's perspective. They looked at the state of experimentation in 10 Finnish IT companies and came to the conclusion that it is not yet mature, as experimentation is rarely systematic and continuous. This is similar to what we have learned from some of our interview participants. However, we were also able to recruit several companies across multiple countries which make heavy use of experimentation. Other notable recent related research has been done by Shahin et al. [55]. Their work focuses on practitioner reports from multiple companies regarding architectural issues of continuously deploying software. This work, which has been conducted in parallel to our study, largely comes to similar conclusions as we do regarding the importance of architecture for implemented experiments.

4. Research Methodology

We conducted a mixed-method study [56] consisting of two rounds of semi-structured, qualitative interviews combined with a quantitative survey. Figure 2 provides an overview of the research methodology. All interview materials and survey questions are part of the paper's online appendix[1]. Further details on the design and execution of our study complementing the information presented here can be found in our case study protocol [57, 58] in the paper appendix. Prior to conducting the initial round of qualitative interviews, we performed a pre-study to identify practices associated with continuous experimentation.

4.1. Pre-Study

Protocol. The goal of the pre-study was to serve as a basis for formulating questions for the qualitative part of our study. As a starting point, we studied Rahman et al. [26], Feitelson et al. [2], Humble and Farley [4], and the Forrester report [27], which we considered standard CD literature at the time we conducted our pre-study (the mapping study by Rodriguez et al [24], which we also consider seminal for the field, was not yet available). In addition we studied multi-vocal literature [59], i.e., unpublished or non-peer-reviewed sources of information usually produced by organizations or practitioners. This included studying tech blogs of industrial leaders such as Facebook [60], Etsy [61], Twitter [62], Google [63], and Netflix [64]. These companies are known for conducting experiments and using highly automated release processes, hence we used their blog posts to supplement the studied academic resources. To avoid potential bias introduced by our selection of blogs and inspired by Barik et al. [65], we then used Hacker News[2] as an additional tool to identify further popular web resources. Articles were found using *hn.algolia.com*, a keyword-based Hacker News search engine. We searched for articles containing the keywords "continuous delivery" and "continuous deployment", which were posted between Jan 1 2011 and Nov 1 2015, and sorted them based on their popularity on Hacker News. For both keywords, we considered the first 80 articles. Our primary focus was on articles containing mainly experience reports, i.e., how companies make use of CD or continuous experimentation in the trenches. We removed those with dead links and those that mainly advertised specific tools. We ended up with 17 (continuous delivery) and 25 (continuous deployment) matching articles. We analyzed the articles based on the usage of CD and experimentation practices, compared them to the findings derived from literature, and created an interview guide divided into five themes: the release process in general, roles and responsibilities, quality assurance, issue handling, and release and experiment evaluation. A full list of articles, the detailed search criteria, and the resulting interview guide can be found in our online appendix.

Threats to Validity. The interview guide and the selection of questions for the qualitative phases of the study might have lead participants to answer towards our possibly biased notion of CD and experimentation (i.e., researcher bias). We mitigated this threat by building a foundation of understanding on the topic that is based on both previous academic work and online articles of well-known industry representatives. Potential bias introduced by our selection of these representatives is mitigated by including further experience reports and articles gathered via a keyword-based search on Hacker News. However, identifying whether a Hacker News article is suitable (i.e., whether it is an experience report or only advertising a service or tooling) introduced a further potential bias that we mitigated by having the authors discuss the relevancy of each article. Another limitation regarding the suitability and validity of the interview questions as a result of the pre-study is that the first author designed all the questions. However, they were rigorously reviewed and verified by the other authors. In addition, some of the questions got improved based on participant feedback during the study.

4.2. Qualitative Interview Study (Interview$_1$)

Protocol. Based on the interview guide generated in the pre-study, we then conducted a first round of interviews. We fostered an exploratory character via a semi-structured interview process. All interviews included the mentioned five themes and discussion of each theme started off with an open question. Except for the first theme, topics were not covered in any particular order, but instead

[1] `http://www.ifi.uzh.ch/en/seal/people/schermann/projects/expstudy.html`

[2] `https://news.ycombinator.com/`

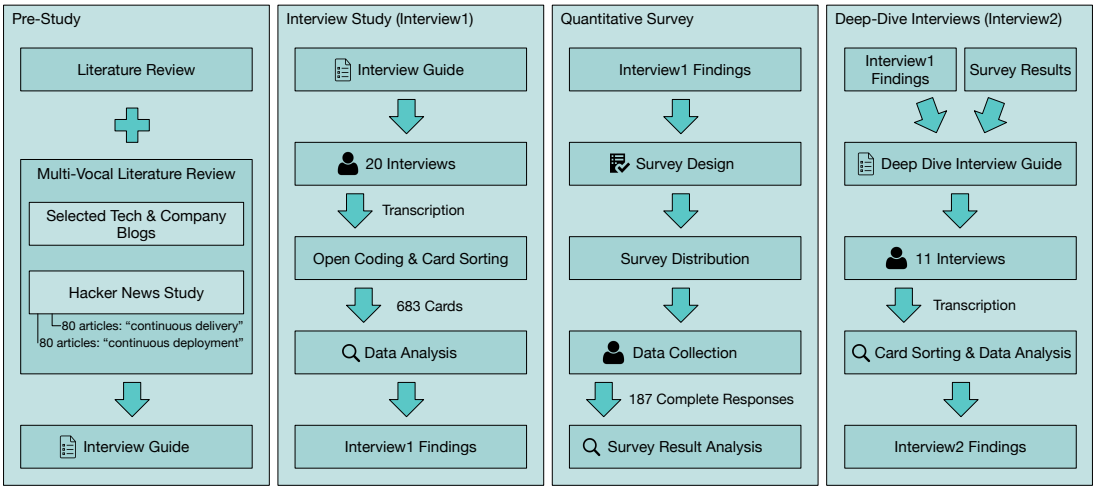

Figure 2: Overview of research methodology consisting of four steps.

followed the natural flow of the interview. In total, the interview guide for this phase consisted of 52 questions. However, we did not ask every single question to each participant. The questions we asked rather depended on the flow of the interview and thus whether certain follow-up questions for the five themes were promising. Both, open and follow-up questions, can be found in our online appendix. The interviews were conducted by the first, the second, and the fourth author, either on-site in the areas of Zurich and Vienna, or remotely via Skype. All interviews where held in English or German, ranged between 35 and 60 minutes, and were recorded with the interviewee's approval.

Participants. We recruited interviewees from industry partners and our own personal networks, and increased our data set using snowball sampling [66], i.e., by asking existing interviewees to put us in contact with further potential interview partners that they are aware of. In total, we conducted 20 interviews in this phase, with developers or release engineers (P1 to P20, one female) from companies across multiple domains and sizes (see Table 1 and Figure 3). These companies range in size from single-person startups to global enterprises with more than 100,000 employees, and are located in Austria, Germany, Switzerland, Ireland, Ukraine, and the US. To ensure a broad understanding of the impact of CD and continuous experimentation on software development, we interviewed practitioners with different levels of seniority (average 9 years, standard deviation 5 years) and different project roles. However, we required that all participants have insights into (technical) details on their company's or project's release process. We primarily selected companies developing Web-based applications, as our pre-study has shown that this is the application model most amenable for continuous experimentation. However, in spirit with the exploratory nature of our study, we also included other application types when companies mentioned their use of CD or continuous experimentation. Although participants *P9*, *P10*, and *P11* are employed by the same company, their teams work on different products utilizing different technology stacks and release processes. Due to the nature of the interviews, some of the questions target personal opinions, while others target the process, team, or even company level. Consequently, when discussing and reporting the results we sometimes refer to the participant's companies or team.

Analysis. The recorded interviews were transcribed by the first two authors. We coded the interviews on sentence level without any a priori codes or categories. The first three authors then analyzed the qualitative data using open card sorting [67] (683 cards in total), and categorized the participants' statements, resulting in the set of findings described in the following. All findings are supported by statements of multiple participants. All selected quotes of interviews held in German were translated to English.

Threats to Validity. For the objectives of this study it was important to recruit interview participants that are approximately evenly distributed between organizations of varying sizes, divergent domains, and backgrounds (years of experience and age of participants). Snowball sampling helped us to increase our sample size. However, a potential disadvantage of this strategy is that it may suffer from community bias, as the first participants are prone to impacting the overall sample. We addressed this threat by selecting study participants purposefully, focusing on practitioners and also reviewed their online profiles, especially for those participants which were suggested to us via snowballing. Further, a potential threat to our empirical findings is that our results are not generalizable beyond the subjects involved in the interviews. We mitigate this effect by employing a mixed-method study validating our interview findings in a more general context using a quantitative survey in the following step. Furthermore, we rely on self-reported (as opposed to observed) behavior and prac-

tices (self-reporting bias). Hence, participants may have provided idealized data about the CD and experimentation maturity of their companies. Furthermore, it is possible that we introduced bias through the mis-interpretation or mis-translation of "raw" results (interview transcripts). To avoid observer bias, these results were analyzed and coded by three authors of the study.

4.3. Quantitative Survey

Protocol. To validate and substantiate the findings from our qualitative interviews on a larger sample size, we designed an anonymous Web-based survey consisting of, in total, 39 questions. Similar to the first round of interviews, we structured the survey into multiple themes: release process in general, software deployment, and issues in production. The survey mainly consisted of a combination of multiple-choice, single-choice, and Likert-scale questions. Although the survey had its focus on quantitative aspects, we also included some free-form questions to gain further thoughts and opinions in a more qualitative manner. Depending on individual responses, we displayed different follow-up questions (i.e., branches in the survey) for the purpose of identifying underlying reasons (e.g., reasons for making use of canary releases, and reasons against). In total we had 7 branches (i.e., 7 mandatory questions) in our survey, thus the number of questions a participant had to answer varied. With this survey design we wanted to avoid presenting a participant with questions that do not make sense based on her previous answers.

Participants. We distributed the survey within our personal networks, social media, via two DevOps related newsletters[3,4], and via a German-speaking IT news portal[5]. As monetary incentives have been found to have a positive effect on participation rates [68], we offered the option to enter a raffle for two Amazon 50$ gift vouchers on survey completion. In total, we collected 187 complete responses (completion rate of 28% out of 667 responses). On average, it took the participants 12 minutes to fill out the survey. The survey was available online for three weeks in February 2016. Survey participants reported an average of 8 years of relevant experience in the software domain (standard deviation 4 years). Similar to the interviews, for some questions we were interested in the development and deployment process on the team or company level. Hence, we sometimes stick to the company level when discussing and reporting results. The resulting participant demographics for the survey is summarized on the bottom part of Figure 3.

Analysis. We analyzed the distributions of responses to Likert-scale, multiple-choice, and single-choice questions. In particular, we have correlated survey responses with the application model (Web-based or other) and the company size, as these two factors have emerged as important factors of influence in the interviews. Further, we coded the answers to open questions in the same style as for the interviews.

Threats to Validity. We advertised our survey over various social media channels to attract a high number of respondents. However, participation in online surveys is necessarily voluntary. Hence, it is likely that the survey has attracted a respondent demography with substantial interest and familiarity with CD and experimentation practices (self-selection bias). Furthermore and similar to our interviews, participants may have provided idealized data about their companies' states on CD and experimentation (self-reporting bias). We piloted the survey with a small initial set of practitioners and gathered feedback to improve the survey before distributing it to a larger community and to avoid potential sources of ambiguity. Similar to our interview transcripts, there is the possibility that we introduced bias through mis-interpreting or mis-translating "raw" results gathered from the free-form questions in the survey.

4.4. Qualitative Deep-Dive Interviews (Interview$_2$)

Protocol. When revisiting our interview and survey findings, we identified the following topics to be of particular interest: (1) experiment design (e.g., metrics, hypotheses, duration), (2) implementation techniques for experiments, and (3) experiment result interpretation. In order to get more profound insights, we defined a set of 32 more detailed questions and conducted a second round of structured interviews. We followed the same protocol as in the initial interview round. Interviews lasted between 20 and 30 minutes and were again recorded with the interviewee's approval. Note that, we did not ask every single question to each participant, as this would have exceeded the targeted time frame. The questions we asked depended on the flow of the interview and thus the different techniques applied by the participant's company or team.

Participants. We again recruited participants from our personal networks and through snowball sampling. In total, we conducted 11 additional interviews with developers or release engineers (D1 to D11) from 9 different companies in various domains located in Germany, Switzerland, the United Kingdom, and the US (see Table 1 and Figure 3). Participants *D4* and *D5*, and participants *D6* and *D11* are employed by the same companies. However, as in the first interview phase, all participants work on different teams, and participants *D6* and *D11* also work on different products. On average, participants of the second round of interviews had 12 years experience (standard deviation 7 years). All of the selected companies for the second round of interviews develop Web-based applications.

Analysis. The recordings of the second round of qualitative interviews were transcribed by the first author. The first three authors again used open coding to categorize

[3] `http://www.devopsweekly.com/`
[4] `http://sreweekly.com/`
[5] `http://heise.de`

ID	Company		Application		Interviewee			Team Size
	Type	Country	App. Type	App. Domain	Role	Experience (in Years) Total	In Company	
P1	SME	AT	Web	Sports News & Streaming	DevOps Engineer	3	3	3–6
P2	SME	AT	Enterpr. SW	Document Composition	Software Engineer	4	4	3–5
P3	SME	CH	Web	Employee Management	Software Engineer	10	5	1–3
P4	SME	CH	Web	Telecommunication	Software Engineer	15	4	3–7
P5	SME	AT	Web	Online Retail	Software Architect	5	5	15–20
P6	SME	AT	Desktop	SharePoint	Software Engineer	4	4	2–7
P7	Corp.	UA	Web	Employee Management	Software Engineer	5	5	4–6
P8	SME	AT	Enterpr. SW	Insurance	Software Engineer	12	12	5–8
P9	SME	CH	Enterpr. SW	E-Government	Solution Architect	13	13	4–6
P10	SME	CH	Web	Mobile Payment	Solution Architect	16	6	60–70
P11	SME	CH	Web	Mobile Payment	Solution Architect	11	4	15-20
P12	Corp.	DE	Web	Cloud Provider	DevOps Engineer	1	1	9–11
P13	Startup	AT	Web	Online Code Quality Analysis	DevOps Engineer	16	1	1
P14	Corp.	IE	Web	Network Monitoring	Public Cloud Architect	10	1	6–8
P15	Corp.	US	Web	Cloud Provider	Program Manager	15	3	8–10
P16	SME	AT	Enterpr. SW	E-Government	Project Lead	15	9	3–7
P17	Startup	US	Web	Babysitter Platform	Software Engineer	4	2	6–8
P18	Startup	US	Web	Event Management	Director of Engineering	5	1	5–7
P19	SME	US	Web	E-Commerce Platform	Software Engineer	5	3	3–7
P20	SME	AT	Embedded SW	Automotive Software	Software Engineer	3	3	3–5
D1	SME	US	Web	CMS Provider	DevOps Engineer	10	1	3–5
D2	SME	DE	Web	Q&A Platform	Head of Development	10	3	4–7
D3	Startup	CH	Web	HR Software	Head of Development	10	7	4–5
D4	SME	DE	Web	Travel Reviews & Booking	Software Engineer	7	2	5–7
D5	SME	DE	Web	Travel Reviews & Booking	Software Engineer	8	2	4–6
D6	Corp.	CH	Web	Telecommunication	Team Lead	5	4	7–9
D7	Corp.	UK	Web	Scientific Publisher	Director of Engineering	9	3	3–12
D8	SME	CH	Web	Network Services	Team Lead	30	3	5–8
D9	Corp.	US	Web	Video Streaming	Head Release Engineering	19	3	5–9
D10	SME	CH	Web	Sustainability Solutions	DevOps Engineer	10	8	1–4
D11	Corp.	CH	Web	Telecommunication	Software Engineer	10	2	5–10

Table 1: Interview study participants of both rounds of interviews

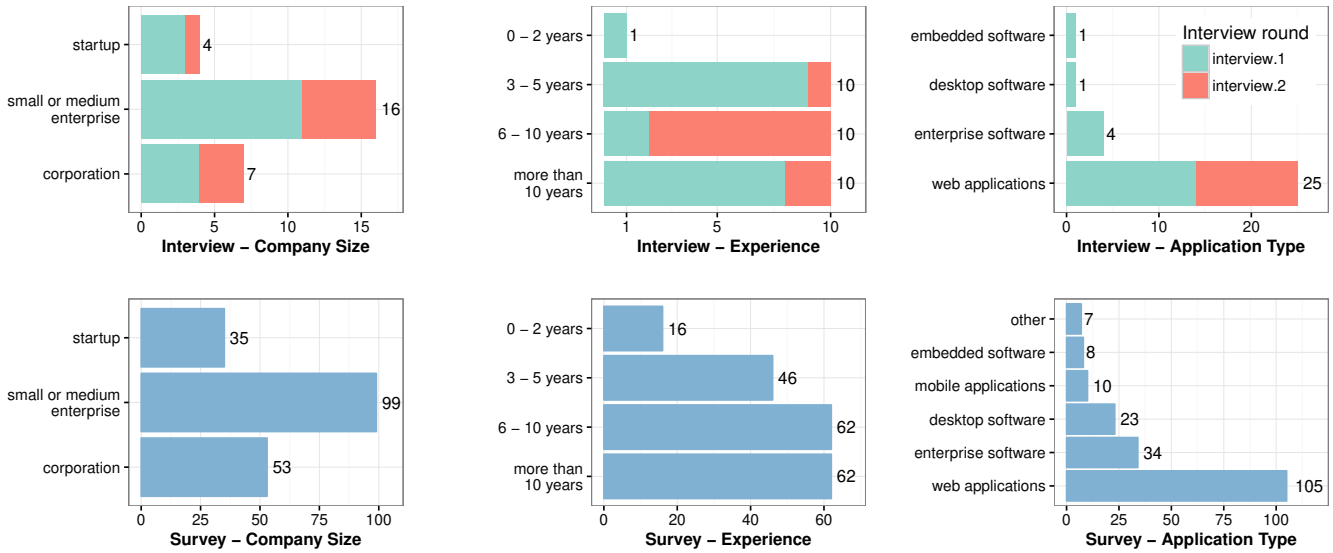

Figure 3: Demographics of interview study participants (top) and survey participants (bottom) subdivided into company sizes (left), experience (center), and application type (right).

the participants' statements and to gather more profound insights into continuous experimentation.

Threats to Validity. For the second round of qualitative interviews we are subject to the same threats to validity as in the first round of interviews that the reader should keep in mind when interpreting our results. We again recruited interview participants that are evenly distributed between organizations of varying sizes, divergent domains, and backgrounds. However, in this phase of interviews we focused solely on companies developing Web-based applications. As we were especially interested in continuous experimentation, we provided potential interview candidates with a brief outline of the goals of our study. While this allowed us to filter for participants that could provide us with useful information, this also introduced a potential threat that they shared information based on what they thought we wanted to know (i.e., hypothesis guessing), or withheld information or opinions that they thought would

be unpopular (i.e., evaluation apprehension) [69]. We mitigated this threat by assuring that both their answers and company affiliation would be anonymized.

5. Practices for Continuous Experimentation

In this section, we cover best practices that facilitate continuous experimentation which emerged from our study. We start with technical practices (e.g., automation, architectural considerations) and move on to more organizational and cultural topics (e.g., awareness, developer on call).

5.1. Technical Practices

Automation and CI. To enable continuous experimentation, companies need to invest in deployment automation. A common implementation in CD projects are deployment pipelines [13, 4]. Such pipelines consist of multiple defined phases a change has to pass until it reaches the production environment. The intrinsic goal behind investments in CD is to increase velocity, i.e., the time needed to pass all the quality gates and approval steps until a change reaches the production environment, while at the same time ensuring that the quality of the resulting product stays high [42]. Recently, there has been a multitude of research works on the challenges companies face on their way adopting CD, including technical and organizational [34, 35, 1], as well as social challenges [37]. Our findings on obstacles regarding deployment automation are in line with existing research, including companies' internal policies (e.g., testing guidelines that are too strict in case of *P4*), or customers which do not appreciate higher release frequencies (e.g., *P9*).

Concerning continuous integration (CI), an often-cited prerequisite for CD and continuous experimentation [4], all but one company have embraced CI. However, CI has been widely covered by recent research. Hence, we omit a more detailed discussion on this topic and refer the reader to existing work (e.g., [14, 15]) covered in Section 3.

Architectural Concerns of Continuous Experimentation. A suitable software architecture has been shown to be essential for experimentation, as it influences both, a company's velocity and release frequency:

> "*It is difficult to release individual parts of the system as dependencies between new code and the system in the back are just too high*" -*P5*

To tackle this problem, *P5* mentioned that in his company they have started migrating from their monolithic application architecture to smaller, independently deployable services (i.e., microservices) [8, 70, 71]. A similar result has also recently been independently reported by Shahin et al. [55]. More generally, we have observed this trend across all our interviewees who use experimentation extensively. All of the companies they work for either have migrated to, or started from scratch with, a microservices-based software architecture. Different parts or functionality of a system are usually developed at a different pace and in different teams, so it comes quite natural that companies favor this option of independently deploying certain parts of their system. Another benefit our interviewees (e.g., *D7*) mentioned is that functionality is implemented with the technology which fits best, and non-monolithic architectures reduce the aversion of experimenting with more recent technology stacks. However, migrating to or designing architectures with many loosely-coupled entities bears its own risks. Suboptimal design decisions (e.g., using a central database for all services) lead to painful releases involving costly coordination among multiple teams whenever database schema changes occur (e.g., *D6*). However, once monoliths are broken down into multiple services (e.g., 70 – 80 services for *D4's* company, hundreds of services in case of *D9*), identifying the root causes of production issues becomes more challenging:

> "*[Root cause analysis] is difficult, and that's one of the main problems we face and we still have to tackle. If there is a severe issue and something is not working, guesswork starts, everyone's asking about reasons and trying things out*" -*D4*

Many teams and services are involved in troubleshooting these distributed problems. Traces of failed requests need to be carefully analyzed, and multiple deployments and their changes and running experiments have to be considered. One approach to tackle this is by forming a separate, centralized team or task force supporting the decentralized service teams.

> "*[...] they will get all the services in that area on basically a Slack channel, and then relevant engineers will start looking at their services and it's like a war room.* " -*D9*

Implementation Techniques. We observed multiple options on how to technically implement continuous experimentation. There is no "one size fits all" solution, and many companies combine multiple implementation techniques.

Feature Toggles. The implementation technique for continuous experimentation that was named most frequently in our study are feature toggles [52, 9]. They are used for canary testing and for gradual rollouts (e.g., *D2*, *D9*), for hiding not yet finished features in production code (e.g., *D7*, *P20*), to bucket users into groups for A/B testing (e.g., *P19*), or for dark launching new functionality (e.g., *D9*). Interestingly, some of our interview participants associated feature toggles with permission mechanisms, e.g., for regulating user access to specific features (e.g., *P9*, *D6*). *D2* appreciate that properly managed and synchronized (e.g., using tools such as ZooKeeper[6]) feature toggles give them more control over their application ecosystem:

[6] `https://zookeeper.apache.org/`

> *"We do [feature toggling] on backend and frontend services, and especially on our iOS and Android apps because of their restricted (app store) release cycles. You want to be sure that if something is wrong, you can turn it off immediately across all frontends." -D2*

As also reported by Rahman et al. [52], our interviewees mentioned technical debt [72] and the additional level of complexity feature toggles add to systems (e.g., *P13*, *D6*) as major drawbacks. As Hodgson [9] stated, feature toggles are easy to use, but they come with a maintenance cost. *D2* mentioned that they reached a point where continuously maintaining and testing 150 feature toggles became infeasible due to state explosion. Issues appeared when someone inadvertently flipped a flag and reactivated dead code. As a consequence, they drastically reduced and limited the number of feature toggles that are allowed to be active at the same time.

> *"I'm not using feature toggles and I don't intend to do so [...] Configuration leads to complexity, and every time you add complexity, you end up with additional complexity when you have to remove it at some point." -P13*

Runtime Traffic Routing. Besides feature toggles, another common implementation technique is runtime traffic routing (e.g., *D2*, *D5*). Depending on request header information (e.g., set cookies, device information), user requests are routed to selected backend instances, and, consequently, to specific versions of the software.

> *"When we could not make it with feature toggles (about 20 – 30% of the cases), we had to think about alternatives. In case of AdSense and Optimizely we set a cookie such that a user always gets the same version." -D2*

A special type of traffic routing that is commonly used among our interview participants' companies are blue/green deployments [13]. They use blue/green deployments mainly for canary testing followed by gradual rollouts. Once the first instance of new version works as expected, the remaining old instances are replaced in a stepwise manner, until a full rollout is reached.

Early Access. A final, relatively conservative, variation of continuous experimentation among our participants is providing specific users or user groups early access to binaries (e.g., *P8*). The main advantage of this model is, unlike for instance traffic routing, that it is not specific to Web-based applications. However, the downside is that the application provider has limited control over their experiments, and cannot, for instance, enforce the usage of the new version for specific users. Further, this experimentation scheme requires substantial manual and administrative effort.

Our interview findings are partially in line with our survey respondents (see Table 2). We use a color coding scheme throughout the tables of this paper in which darker cell background colors emphasize higher percentage values. Due to our focus on companies offering Web-based products in the qualitative parts of our study, we had only one company (*P8's* company) providing their software in form of binaries, as opposed to our survey participants with 29%. Regarding our survey participants, feature toggles are especially used by companies providing Web-based products (45%), while they are less frequently used for other application models (25%). While traffic routing is also frequently used for Web-based products (45%) among our survey participants, it is less important in other application types (12%), in which pre-access to binaries is more common (47%).

	all n=70	**Web** n=38	**other** n=32	**start.** n=8	**SME** n=43	**corp.** n=19
other	6%	8%	3%	12%	5%	5%
permissions	17%	18%	16%	38%	16%	11%
dont' know	20%	13%	28%	12%	21%	21%
binaries	29%	13%	47%	12%	33%	26%
traffic routing	30%	45%	12%	38%	23%	42%
feature toggles	36%	45%	25%	50%	35%	32%

Table 2: Implementation techniques in use for continuous experimentation (multiple-choice).

Monitoring. An effect of highly automated pipelines is that not only new features reach production faster, but so do bugs. While delivery pipelines typically consist of a number of automated or manual quality checks, bugs are bound to slip through on occasion. This changes the way how companies have to deal with issues:

> *"I think the faster you move, the more tolerant you have to be about small things going wrong, but the slower you move, the more tolerant you have to be with large change sets that can be unpredictable." -P18*

Highly automated pipelines allow companies to fix those small issues fast. Monitoring is a prerequisite for keeping developers aware of events in production environments. With continuous experimentation, the importance of monitoring applications even increases. Monitoring is not only used to determine if everything runs as expected (i.e., through health checks), but also to support rollout decisions (e.g., increase traffic assigned to a canary release) and decide about the continuation of ongoing experiments and the outcome of completed experiments (e.g., determining the outcome of an A/B test).

> *"The decision whether to continue rolling out is based on monitoring data. We look at log files, has something happened, did we get any customer feedback, if there is nothing for a couple of days, then we move on." -P16*

Interview participants mentioned that they do not only rely on monitoring data to identify runtime issues, but also take customer feedback, for instance provided via bug reports, into account. This was also supported by our survey results. Customer feedback (85%) and active monitoring (76%) are both widely used among survey respondents (see Table 3). For Web-based applications, monitoring and customer feedback are in balance, while for other application types, customer feedback (90%) is dominant (67% monitoring). This is not surprising, as monitoring Web-based applications is technically easier than for other applica-

tion models, and supported by existing Application Performance Monitoring (APM) tools, such as New Relic [73].

	all n=187	Web n=105	other n=82	start. n=35	SME n=99	corp. n=53
don't know + other	4%	2%	6%	3%	5%	2%
monitoring	76%	83%	67%	89%	72%	75%
customer feedback	85%	81%	90%	80%	88%	83%

Table 3: How issues are usually detected (multiple-choice).

5.2. Organizational and Cultural Practices

Awareness. Awareness refers to activities that foster transparency of the development and experimentation process for every stakeholder (e.g., developers, testers, operations). Similarly to monitoring, awareness is becoming even more important once continuous experiments are conducted. Multiple deployments and experiments conducted at the same time can negatively influence data collection and statistically robust analysis, i.e., correctly identifying and dealing with the noise induced by concurrent experiments. Consequently, it is important that developers, release engineers, and other stakeholders stay informed. We distinguish between awareness throughout the development process, and during experimentation. The former typically covers tooling that tracks status or progress of features through tasks or tickets (e.g., Pivotal tracker). The latter involves various ways of informing other teams about experiments being conducted, e.g., internal wiki or blog posts (*D1*), e-mail notifications (*D9*), or meetings of product owners and team leads (*D2*). Combined solutions involve online dashboards, or public screens in the office, which display information such as build status, test results, or production performance metrics. Another way to promote awareness and transparency is through signals sent in the form of asynchronous communication tools that are integrated with the team collaboration chat tools, such as Slack or HipChat [74].

Developer on Call. Interviewees agree that the notion of developer on call, i.e., that a developer needs to be available to provide operational support after a release, has become a widely accepted practice in their organization. This was not only the case for companies following a service-based architecture, where being responsible as a team for your own services comes naturally, but also for other companies we interviewed. In case of issues, developers know best about their changes and can help operations to identify the problem faster and contribute to the decision about subsequent actions. Additionally, *P16* also specifically mentions a learning effect for developers:

> "*Developers need to feel the pain they cause for customers. The closer they are to operations the better, because of the massive learning effect.*" *-P16*

This practice is strongly related to DevOps and emphasizes a shift in culture that is currently taking place. Traditional borders between development, quality assurance, and operations seem to vanish progressively. This addition of responsibility could lead developers to writing and testing their code more thoroughly, as some participants indicated:

> "*If you don't have enough tests and you deploy bad code it will fire back because you would be on call and you have to support it*" *-P14*

Some participants (e.g., *P7*) mention that their companies avoid the additional burden of keeping developers on call on weekends by releasing only during office hours. However, for many companies and domains, deployment weekends are a business necessity (e.g., *P9*). Others follow a more pragmatic process with a clear handover of responsibility. For instance, at *D5's* company, developers provide a manual containing step by step descriptions for operations on how to act in certain circumstances (e.g., rollbacks, flipping a feature toggle to turn off the experiment).

Our survey confirmed these findings (see Table 4). The majority of survey respondents stated that developers never hand off their responsibility for a change. When comparing company sizes, developers are on call particularly at startups (74%), but even in larger corporations this concept is applied frequently (45%). While in SMEs and corporations (23%) developers hand off their responsibility directly after development, this is almost never the case for startups (3%).

	all n=187	Web n=105	other n=82	start. n=35	SME n=99	corp. n=53
don't know + other	4%	2%	5%	3%	1%	8%
preproduction	9%	10%	9%	9%	8%	11%
staging	12%	15%	9%	11%	12%	13%
development	19%	12%	28%	3%	23%	23%
never	56%	61%	50%	74%	56%	45%

Table 4: Phase in the release process after which developers typically hand off responsibility for their code (single-choice).

Decentralized Teams and Consultants. Many interview participants are not only supported by central teams providing infrastructure (e.g., deployment pipelines, containers with pre-configured monitoring) and tooling, but also by a range of consulting teams. In companies that adopt microservices, teams developing functionality are autonomous in most of their decisions, including experimentation. However, not all teams are staffed with experts in all relevant fields to either conduct or interpret experiments (e.g., data scientists, DevOps engineers). These teams can request support from centralized teams, e.g., for identifying the right set of metrics and thresholds to assess a service's health state (e.g., *D9*, *D1*). We further observed that tooling and infrastructure provided by a centralized team increase technology homogeneity, since they not only provide, but also maintain, standard tools:

> "*[...] you are allowed whatever tool you want. The interesting thing is, [...] teams are not required to use [tool name] if they don't want to, but everyone uses it*" *-D9*

Teams using their own technology stacks are required to maintain them, leading to additional effort. Further, the service team is held responsible when their non-standard tools fail or lead to other issues.

6. Conducting Experiments

After covering common practices, this section focuses on how companies actually conduct continuous experiments. A central aspect that emerged from our study is that there are fundamentally two classes of experiments, namely experiments conducted to identify and mitigate the impact of software regressions, such as functional bugs that evaded detection in the delivery pipeline, performance regressions, or scalability issues (*regression-driven experiments*) and experiments conducted to evaluate different software design or implementation decisions from a business perspective (*business-driven experiments*). While superficially similar on a technical level, different concrete practices are typically used to implement those classes of experiments. Business-driven experiments are primarily conducted using A/B testing. For regression-driven experiments, multiple techniques are in use, including canary releases, dark launches, and gradual rollouts. We summarize the main characteristics, differences, and commonalities of these classes of experiments in Table 5.

6.1. Regression-Driven Experiments

This variant is about mitigating technical risks and verifying the correct functioning of a new version or feature. Regression-driven experiments are used to detect functional problems that slipped through unit or integration testing, performance regressions, or new features that do not scale to production workloads.

> "*Even though [a new feature is] tested in test, it's still the data combinatorics in production are so vastly different than we can simulate in test that in some cases we do find issues in production.*" *-D9*

Such production "health checks" are implemented in various ways and on differing scales. A commonly used practice among our interview participants are canary releases. Release engineers either make use of them for all changes, or, more commonly, use this practice for specific changes that are considered particularly critical. A typical use case is scalability testing in Web-based applications (e.g., *P4*, *D2*).

> "*[We use canary releases] especially in those cases when we have concerns how it would scale when all users get immediate access to this new feature.*" *-P4*

Our survey has shown that 63% of practitioners are not using any variant of regression-driven experimentation (Table 6). Consistent with our interview results, this flavor of experimentation – among those that actually make use of it – is not bound to companies developing Web-based applications. There is no significant difference in our survey responses in terms of its adoption between developers of Web-based applications and others. However, for developers using other application models, partial rollouts usually come in the form of simple pilot or early access phases. These are usually manually-administered with hand-picked friendly customers (e.g., companies of *P8*, *P9*, *D3*). This concept is similar to pre-release versions (e.g., alpha, beta, RC) sometimes used in desktop and enterprise software. Such early-access canaries are typically not systematically monitored and experiment outcomes are determined primarily by analyzing user feedback.

Dark or shadow launches, as pioneered by Facebook, are rarely used among our interviewees. Only *D9* conducts dark launches in a similar fashion as described by Facebook, by implementing and controlling experiments using feature toggles. *D1* mentioned that they do not have the necessary scale for it, and *D2* does not see a pressing need. *D5*, however, occasionally conducts a simplified version of dark launches:

> "*We do have a procedure such that as long as a service [version] is not effectively enabled in production we push every feature branch to prod, thus we can ensure that it runs as we expect [...] including [real or generated] traffic would be the next logical step.*" *-D5*

Metrics. In case of canary releases, measured metrics consist of standard application (e.g., response times) and infrastructure (e.g., CPU utilization) metrics. This is consistent with our results from a previous study [70]. Interviewees did not have strict rules on what to monitor, nor do they have access to clear thresholds or tests that help them assess whether specific monitoring data should be considered "healthy" for a given application. Instead, practitioners conduct health assessments iteratively and primarily based on intuition. If a metric value appears problematic (e.g., appears to be visually different in a dashboard), they take action based on informal past experience rather than well-defined processes and empirical data. This is consistent with our experiences in earlier studies [29, 75]. If formal thresholds are used, they are often based on historical metrics gathered from previous releases. Even though a minority in our study population, some interviewees (e.g., *D2*, *D4*, *D9*) also used *a priori* defined metrics and thresholds.

> "*On a low level basis, [...] [we] basically do an apples to apples comparison for about 2000 metrics, so every team is kinda free to pick their own. [...] they are looking for deviations [...] if you spin up version 2, it does a comparison and then you can basically say what the variance is allowed to be.*" *-D9*

Notifications are typically sent automatically if the data shows any (negative) deviations from the baseline version. *D5* mentioned that setting concrete thresholds is tricky and often leads to false alarms. Hence, they refrain from setting specific thresholds.

Responsibility. In microservices-based architectures, which many of our interview participants use extensively

	Regression-Driven Experiments	Business-Driven Experiments
Main Goals	Mitigation of technical problems (e.g., related to bugs or performance regressions), conducting health checks, testing scalability on production workload	Evaluation from a business perspective of new features or different implementation decisions (do customers appreciate the change, is it in line with monetary incentives and company goals?)
Common Practices	Canary releases, dark launches, gradual rollouts	A/B testing
Used Metrics	Typically multiple application and infrastructure level metrics (e.g., response time) in combination with simple-to-measure business metrics	Primarily business metrics, sometimes combined with small selection of application metrics
Data Interpretation	Often intuitive and based on experience, less process driven (do metrics "seem higher than before"?)	More statistically rigorous hypothesis testing based on carefully selected metrics
Experiment Duration	Minutes to multiple days	Often in the order of weeks (see also Kevic et al. [44])
Selection of Target Users	Often small scoped (e.g., small percentage of users, user groups, regions), sometimes gradually increased until full rollout	Two or more groups (percentage of user base, user groups, regions) of same size, constant size during experiment
Responsibility	Siloization, single team or developers	Multiple teams and services involved, requires coordination, awareness, and commitment across team borders
Impl. Techniques	Feature toggles, dynamic traffic routing, distribution of different variants in form of binaries	
Main Obstacles	Architecture, limited number of users, missing business value or not worth investments, lack of expertise	

Table 5: Summary and comparison of regression-driven and business-driven experiments.

	all n=187	**Web** n=105	**other** n=82	**start.** n=35	**SME** n=99	**corp.** n=53
for all features	18%	15%	22%	6%	22%	19%
for some features	19%	21%	17%	17%	21%	17%
no experimentation	63%	64%	61%	77%	57%	64%

Table 6: Usage of regression-driven experimentation (single choice).

(*P10*, *P12*, *P14*, *P15*, *P19*, *D2*, *D4*, *D5*, *D7*, *D9*), regression-driven experiments are often characterized by "siloization". Teams responsible for a service decide when and how long to conduct experiments on this service. Moreover, it is typically the task of the team to interpret monitoring data collected during the experiment. However, not all teams have the necessary data science or domain knowledge to do so with confidence. Hence, centralized support teams are sometimes available that help identify metrics to look after, interpret collected data, and identify issues causing experiments to fail (e.g., *D1*, *D4*, *D9*). In other companies (e.g., *P4's* company), conducting experiments is a shared task between release engineers, team leads, and operations, which is outside the traditional microservice team structure.

Duration and User Selection. The duration of canary tests, dark launches, and gradual rollouts varies from few minutes (e.g., in case of *D7*, whose team conducts very short-term 5-minute health checks) to multiple days, but rarely takes longer than two weeks. The end of the spectrum includes those companies rolling out on a data center level (e.g., companies of *P12*, *D8*) or directly contact their customers for feedback (e.g., through early access phases in case of *D3* and *P8*). The amount of, and which, users are considered for an experiment depend on a new feature's complexity, i.e., the more critical a feature, the higher the risk, thus the smaller the scope of the experiment initially. User selection varies and involves random selection on user traffic level, specific user groups (e.g., role, device), or entire regions and countries. Some companies (e.g., of *P16* or *D1*) apply further risk mitigation strategies by following an "eat your own dog food" [76] approach. That is, they are rolling out and testing new versions of their software internally first before rolling out to external customers.

6.2. Business-Driven Experiments

The primary purpose of business-driven experimentation, most commonly associated with A/B testing, is to evaluate the business value of specific features, implementation decisions, or products. Prerequisite for business-driven experimentation is that the versions under test are technically sound. In our study, the central system-under-test for this type of experiments were user facing frontends. A special case was the company of *D5* that relied on A/B testing also for their migration from a monolithic to a microservices-based architecture. The company of *D7* sometimes conduct, as they called it, "fake A/B tests", in which they were interested in the demand for a certain feature without actually implementing it due to high costs and unknown demand. They integrated a mockup into the user interface and kept track of user interactions.

> "*We used it as our decision basis, in that mentioned case we implemented the feature because data have shown that it generates more downloads and thus more money*" *-D7*

23% of our survey respondents have adopted A/B testing. Interestingly, this practice is not only bound to companies developing Web-based applications, even though they still represent the majority with 63% of A/B test users in our survey. Consistent with our interviews, evaluating changes in the user interface is the most common

use case (88%) in our survey, but backend features are also A/B tested by 44% of the respondents.

Metrics. Due to their higher strategic importance, decision making in business-driven experiments tends to be governed less by intuition and experience, and more by statistically sound data analysis. Companies more often start experiments with clearly defined hypotheses, deciding a priori about what to expect (i.e., metrics and deviations), which users to invite or select, and how long the experiment should take. Our interviewees often had a selection of domain-specific key performance indicators (KPIs) they looked at specifically throughout those experiments, such as conversion rates or sales figures:

> "*It was about evaluating KPIs, how did they perform in both groups, what did we expect. Prerequisite is that you have to ensure during development that you can measure those metrics later on.*" -*D2*

Responsibility. Business-driven experiments often involve more than a single team. For instance, frontend functionality leverages multiple backend services, thus coordination and commitment among all teams along the call path is required. Teams need to make sure that multiple experiments, both regression- and business-driven, do not negatively influence each other. Some companies (e.g., of *D2*) only allow exactly one experiment being conducted for a single part of the application (e.g., frontend site), while others (e.g., of *D1*, *D9*) tackle this problem through long test durations and large sample sizes, and treat other experiments simply as noise:

> "*There is the ability to see if it affected it, but I dont think we necessarily pay too much attention. [...] overall, A/B tests run for a long time, I think they evaluate this as noise*" -*D9*

Experiment data interpretation requires substantial expertise in statistics and data science. Interpretation is either a shared task (e.g., *D1*), or carried out by single team members, often product owners of frontends (e.g., *D2*, *D4*).

Duration and User Selection. The exact duration of business-driven experimentation varied, but was typically in the area of 4 to 6 weeks for our interviewees. Experiment durations are dependent on getting enough data to allow for statistical significant conclusions and to deal with fluctuations:

> "*Feature performance varied on a daily basis, could be different on day three than on day four, that's why we take enough time to [collect data and] draw valid conclusions.*" -*D2*

Similar to regression-driven experiments, user selection strategies vary, and can include random sampling, specific roles or user groups, and regions or countries. Moreover, the concrete user selection strategy depends on the actual feature being tested, and may require coordination with marketing and product development (e.g., in case of *P17*) as well.

In terms of size of test and control groups, we identified different approaches. *D2's* company uses the same sizes each time for test and control groups to facilitate data interpretation. 1% of the user traffic is used as test group for the new feature, and 1% of the user traffic as control group. The remaining 98% get the same version as the control group without being tracked. *D1* mentioned that it depends on the teams experience, some conduct 50:50 scale experiments, others start with 2% versus 98% of user traffic.

6.3. Obstacles of Continuous Experimentation

We now report on the main problems and obstacles to adopting continuous experimentation, both of the regression- and business-driven variety. For the 63% of respondents that are not actually using any variation of regression-driven experiments, the largest obstacle is a software architecture that does not easily support experimentation. This was particularly evident for SMEs and corporations, and for companies that develop Web-based products (64%, versus 48% for others). It is likely that this is because most Web-based products in these domains are still deployed as monolithic 3-tier applications. For startups, software architecture is slightly less of a concern. However, startups often do not have a sufficiently large customer base to warrant regression-driven experimentation. This is linked to a third, similar problem preventing the adoption of this type of experiments – some teams or companies simply do not see any business value in conducting them. Interestingly, lack of expertise was only seen as a minor barrier for adoption, given by 26% of respondents overall. A summary of the main reasons against adopting regression-driven experiments is shown in Table 7.

	all n=117	**Web** n=67	**other** n=50	**start.** n=27	**SME** n=56	**corp.** n=34
other	18%	1%	10%	7%	4%	6%
lack of expertise	26%	27%	24%	15%	34%	21%
no business sense	39%	39%	40%	41%	36%	44%
number customers	39%	46%	30%	56%	38%	29%
architecture	57%	64%	48%	44%	66%	53%

Table 7: Reasons against conducting regression-driven experiments (multiple-choice).

A summary of the main reasons against business-driven experiments as resulting from our survey is given in Table 8. Unsurprisingly, and similarly to the obstacles for regression-driven experiments, for those 77% of participants that are not making use of A/B testing, the biggest challenge is a software architecture that does not support running and comparing two or more versions in parallel. Unsuitable software architectures are mainly a problem for SMEs and corporations, while for startups a small user base is seen as a major obstacle. This also was an issue that emerged from our interviews:

> "*We only have around 130 customers, it is actually easier to just talk to everybody.*" -*P18*

Once enough data points are collected to ensure statistical power, expertise is needed to analyze and draw

valid conclusions. However, a lack of expertise was only mentioned by a minority of respondents (15%) as a problem. Interestingly, many companies report that they do not have the features for which it would be worth conducting A/B tests. A similar theme has also emerged in the interviews. The return on investment, both financial and time, of creating and/or setting up appropriate tooling would be just too low. This was mentioned by 33% of our survey participants. While limitations because of internal policies are minor factors for startups, for corporations this represents a strong barrier.

	all n=144	**Web** n=78	**other** n=66	**start.** n=25	**SME** n=74	**corp.** n=45
other	6%	4%	8%	4%	1%	13%
don't know	6%	5%	6%	4%	7%	4%
lack of knowledge	15%	19%	11%	12%	15%	18%
policy / domain	21%	14%	29%	12%	22%	24%
number of users	28%	32%	23%	44%	27%	20%
investments	33%	35%	30%	44%	31%	29%
architecture	50%	53%	47%	40%	59%	40%

Table 8: Reasons against conducting business-driven experiments (multiple-choice).

6.4. Summary

Having covered the two flavors of continuous experimentation that emerged from our study, we now want to summarize the usage of continuous experimentation practices among our interview participants (i.e., including both interview rounds Interview$_1$, and Interview$_2$). Table 9 provides an overview of the prevalence of microservices-based architectures, the usage of implementation techniques (i.e., feature toggles, traffic routing, and early access to binaries), whether developers are "on call", and finally, whether development teams are supported by decentralized teams and consultants. Besides those practices, Table 9 also depicts if and of which classes of continuous experimentation (i.e., regression-driven and business-driven) the company or team makes use of. For each participant in our interview studies, we provide a simple mapping whether the participant's team uses (turquois), does not use (white), or partially uses (color graded turquois) a respective practice or type of continuous experimentation. Partial usage means that the respective company or team does have concrete plans to use a practice or is currently in the process of migration (e.g., moving from a monolithic towards a microservices-based architecture).

Developer on call is a widely accepted practice among our interview participants, while decentralized and consulting teams are especially common in larger organizations. Feature toggles and traffic routing are the typical implementation techniques for continuous experimentation. However, although being a niche practice, some of our interview participants prefer a more conservative approach of providing certain users early access to the newest binaries. We also see that microservices-based architectures are strongly represented in those companies making extensive use of either regression-driven or business-driven continuous experimentation. Among our interview participants, regression-driven continuous experimentation is more common than business-driven continuous experimentation. However, four companies do have concrete plans for conducting business-driven continuous experimentation.

7. Implications

We now discuss the main implications of our study. We focus on the underlying problems and principles we have observed, and propose directions for future research.

Architectural support for experimentation. As discussed in Section 6, a (legacy) system architecture is a dividing barrier between companies that do and those that do not adopt continuous experimentation. Such an architecture makes advanced practices, such as canary releases or A/B testing, hard to implement. We have observed that applying feature toggles (see Section 5) to circumvent architectural limitations for implementing experimentation comes at the price of increased complexity, which negatively affects maintainability and code comprehension. Moreover, as reported by Rahman et al. [52] they introduce technical debt. Microservices, or other architectural models that foster independently deployable components or services, are a promising enabling technology to ease experimentation, but the community is currently lacking formal research into the tradeoffs associated with such architectural styles. For instance, we have observed that practitioners currently lack means to decompose an application into microservices in the first place, or identify which microservice is causing a runtime issue along the call path. Further, more studies are needed to assess the suitability of microservices for various continuous experimentation practices.

Modeling of variability. Related to the previous implication, the results reported in Section 5 imply that practitioners currently struggle with the complexity induced by feature toggles. Hence, it can be argued that more research is needed on better formalisms for modeling the software variability induced by feature toggles, as well as for their practical implementation without polluting the application's source code with release engineering functionality. There has been a multitude of research around variability, i.e., how software can be adjusted for different contexts (e.g., Galster et al. [77], Capilla et al. [78]). We suspect that concepts such aspect-oriented software development [79] and (dynamic) product line engineering [80] could serve as useful abstractions in the domain of continuous experimentation. However, their usage did not emerge in our study even though these techniques have been available for years.

From intuition to principled decision making. In Section 6, we have observed that many release engineers are mostly going by intuition and previous experience when defining metrics and thresholds to evaluate the success of regression-driven experiments. Similarly, which

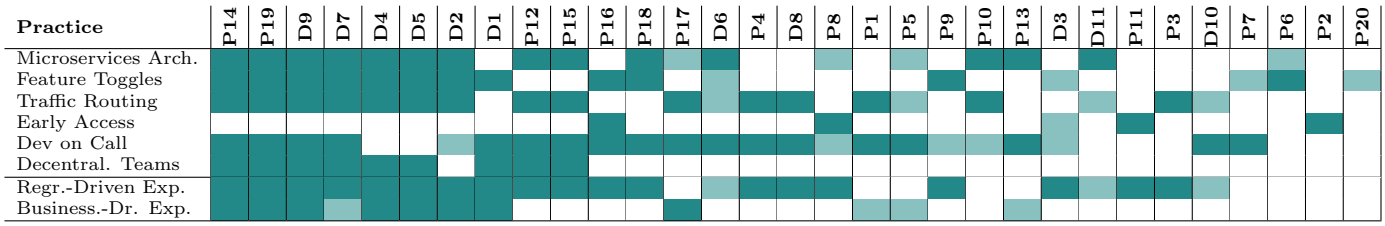

Table 9: Usage of continuous experimentation practices by our interview participants.

features to conduct canary tests on, or which (fraction of) users to evaluate, is rarely based on a sound statistical or empirical basis. Hence, research should strive to identify, for various application types, the principal metrics that allow for evaluating the success of an experiment, and identify best practices on how to select changes that require experimentation. Further, robust statistical methods need to be devised that suggest how long to run at which scope (e.g., number of users) to achieve the required level of confidence. A main challenge for this line of research will be that release engineers cannot generally be expected to be trained data scientists. This is particularly true for smaller companies, for which release decision making needs to remain cost-efficient and statistically sound on a small sample size.

The many hats of developers. An underlying theme of our study results is that developers in those companies that use models such as DevOps, developer on call, microservices, or continuous experimentation, often need to juggle their core task of writing code against many other responsibilities, including operations support, release planning, and data analysis—often under considerable pressure to move fast [3]. While we have observed that some companies provide central support teams (e.g., through dedicated data science consultants [81]), in many companies the teams need to acquire the necessary expertise to handle these new job aspects themselves. However, not only the software developer's role is subject to change in the context of CD and continuous experimentation, but also the software architect's. As reported by Hohpe et al. [82], software architecture has also become broader and more complex, requiring practitioners to steadily keep informed about new technology such as microservices-based architectures. Designing complex systems is more than leveraging object-oriented design skills, it involves leading, mentoring, and conveying complex concepts in approachable terms. Follow-up studies will be required that address these changes in the job profile of software developers and architects.

8. Conclusions

We report on an exploratory, yet systematic, empirical study on the practices of continuous experimentation. Continuous experimentation guides development activities based on data collected on a subset of online users on a new experimental version of the software. As continuous experimentation is especially amenable for Web-based applications, we primarily selected developers or release engineers from companies developing Web-based applications for the qualitative phases of the study. The insights provided by our study help to understand the state of practice in this field, how companies use experimentation and which challenges they face when adopting it. First, many companies practice it as an experience-driven "art" with little empirical or formal basis. Our study suggests that foundational support is needed for moving towards principled approaches for release decision making. Second, small and independently deployable services (e.g., microservice architectures) have emerged as a enabling technology, named by all our interviewees who heavily use experimentation techniques. However, we found that guidelines are required on how to decompose (monolithic) applications and migrate to such microservices-like architectural styles. And third, the advent of experimentation and continuous deployment processes has led to a shift in responsibilities. Developer on call policies have become widely accepted, in which developers are not only responsible for the code they write, but also decide in collaboration with their team which experiments to conduct and which metrics to consider for evaluation.

Once a more principled approach for release decision making is established, we envision this to lead to well-defined, structured continuous experimentation processes implemented in code (i.e., Experimentation-as-Code), analogously to the already established concept of Infrastructure-as-Code [83]. Such Experimentation-as-Code scripts can be structured into multiple phases with clearly specified gateways and repair actions. This will not only provide means for further automation, but also facilitate the documentation, transparency, and even formal verification of experimentation processes. We have already proposed initial steps into this direction as part of our proof-of-concept system Bifrost [53].

Acknowledgments

The authors would like to thank all interview and survey participants. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave), and from the Swiss Na-

tional Science Foundation (SNSF) under project name "Whiteboard" (SNSF Project no. 149450).

Appendix - Case Study Protocol

For planning, conducting, and reporting our study, we followed the case study protocol proposed by Brereton et al. [57]. We further considered the guidelines reported by Runeson et al. [58]. The following sections provide details on study design, case selection, data collection, and analysis.

1. Background

The goal of our study was to identify (1) the principles and practices that enable and hinder organizations to leverage continuous experimentation, and (2) how companies use experimentation and how their techniques differ. In Section 3, we provide details on related work in the area of release engineering and list open issues we tried to address within the context of our study. Prior to starting the data collection process of our study, we conducted a literature review (i.e., our pre-study) to identify typical practices associated with continuous experimentation and derive a questionnaire for the first round of qualitative interviews (see Section 4.1 for details).

2. Design

Our study was designed as an embedded, multiple-case study. We followed a multiple-case design as we are interested in the state of practice in continuous experimentation. Rather than limiting our data collection to a single case study company, we aimed for a more comprehensive view on the field. We interviewed 31 developers or release engineers from 27 companies. The object of study was the release process of the participating companies. Depending on the size and the domain of a company it could be the case that multiple (different) release processes are in place (e.g., for different products, or projects). Within the context of our study, we focused on those processes our participants are associated with, i.e., the product or project they are working on. In three cases during the qualitative phases of the study we collected data from within the same company. For these cases, we ensured that data on projects or products with different release processes were collected. We chose an embedded study design since we are not only integrating data from multiple cases, but also analyze multiple embedded components from each case, i.e., multiple units of analysis. We further position our study as exploratory [58], as it sought to generate new insights, and we adopted a "soft" case approach according to Braa and Vidgen [84] as our research outcome was about gaining understanding. We complemented our qualitative data by conducting a quantitative online survey.

An important step in our study was the literature review (i.e., pre-study) to get a notion of the practices associated with continuous experimentation and to serve as a basis for developing a (first) questionnaire. Section 4.1 provides details on the pre-study, including the considered related research and multi-vocal literature, and the search criteria used. The literature review was not systematic (SLR). Rather, given the exploratory character of our study, we sought to identify a set of key concerns or themes (i.e., forming a theoretical framework) that are important considerations when reviewing the state of practice in continuous experimentation. These key concerns further establish the boundaries of our study and directly link to the units of analysis (i.e., release process, roles and responsibilities, quality assurance process, issue handling process, experimentation process, experimentation design, experimentation implementation techniques, and experiment result interpretation) and consequently, support us in answering our research questions introduced in Section 1.

2.1. Interview Study 1 - Key Concerns

In the following, we will briefly describe the key concerns and themes (i.e., units of analysis) covered in the first qualitative phase of our study.

Release process in general. The goal is to analyze the single phases (e.g., building, testing, deploying) a (code) change has to pass through once a developer pushes the change to the version control system triggering the release process. This allows us to get a first overview of the release process, whether development, quality assurance, and operations tasks are tightly connected or strictly separated within a company, and how automated the entire process is.

Roles and responsibilities. This concern sheds light on the various stakeholders involved in the software release process. Questions covered within this theme involve, amongst others, who decides to ship a certain feature, who is responsible for problems that might appear, are developers required to stay on call, and how aware are the various stakeholders about ongoing and upcoming releases and experiments.

Quality assurance. Within this theme we are interested in details how a company ensures software quality. This involves analyzing whether quality assurance is separated into multiple stages within a so-called deployment pipeline, whether manual approving is necessary in between, and whether builds happen exactly once throughout the release process.

Issue handling. This concern deals with the process of handling issues, whether problems detected in the production environment are treated different than other issues and how those issues are typically detected (e.g., monitoring measures in place), by whom, and how long does it take to fix them.

Release and experiment evaluation. This concern should provide us insights into the experimentation process of a company or project. It unveils how companies keep track of experiments, whether there are strictly defined processes for testing new features on a small fraction

of the user base, and how do the various stakeholders involved interact with each other.

2.2. Interview Study 2 - Key Concerns

In order to get more profound insights into the experimentation processes, we divided the release and experiment evaluation concern into three more detailed key concerns for the second round of interviews that are briefly sketched in the following.

Experiment design. This key concern covers the topic of how companies plan and design experiments. It helps us to determine whether experimentation follows a strict process (e.g., defining hypotheses, pre-selected set of metrics to monitor) or is more driven by the developer's gut-feeling and who is typically involved and responsible when designing and planning experiments.

Implementation techniques. The theme of implementation techniques covers the technical aspect of continuous experimentation. It sheds light on the various techniques (e.g., feature toggles, traffic routing) used for different types of experimentation (e.g., canary releases, A/B testing) and how these are combined.

Experiment result interpretation. Experimentation is all about data collection and data interpretation. We are interested in how companies interpret the collected data, in which intervals, and who is responsible for it.

3. Case Selection

Ideally, the cases (i.e., companies or projects in our study) should be selected intentionally and the units of analysis should have variation in their properties such that the application of data analysis methods reveal new insights. In our study, the recruited companies range in size from single-person startups to global enterprises. For the qualitative phases of our study, we selected companies or projects across multiple different domains (see Table 1). We primarily selected companies or projects developing Web-based applications, as our pre-study revealed that this is the application model which is most amenable for continuous experimentation. However, in spirit with the exploratory nature of the study, we also included other application types when our contacts mentioned their use of CD or continuous experimentation.

4. Case Study Procedure and Roles

The design of our study did not require direct access to specific company or project data (e.g., documentation, source code, test reports). The first round of interviews were conducted by the first, the second, and the fourth author, either on-site in the areas of Zurich and Vienna, or remotely via Skype. The deep-dive interviews (i.e., second round of interviews) were conducted by the first and the second author, either in Zurich, or remotely via Skype. The design of the quantitative survey involved all authors and the survey was hosted on the survey platform Typeform[7].

5. Data Collection

When starting the study, it was not decided how many iterations (i.e., steps) should be conducted. The initial design considered a single round of qualitative interviews and a quantitative online survey. To get more profound insights, we conducted a second round of interviews after the survey phase. Finally, data was collected using two rounds of interviews combined with a quantitative online survey (i.e., data and methodological triangulation). Both data collection techniques include the direct involvement of software developers or release engineers, i.e., first degree contact according to Lethbridge et al. [85]. The interviews with 31 developers of 27 companies are the primary source of information within the context of the study as much of the knowledge that is of particular interest (e.g., current issues with the release process) is not available anywhere else than in the minds of the interviewed participants. The quantitative survey was used to validate and substantiate the findings from the qualitative interviews.

5.1. Interviews

Interview design. Both rounds of interviews followed the same design. Based on the findings of the pre-study (i.e., key concerns and themes) an interview guide was generated that we used to conduct the first round of interviews. For the second round of interviews, we created a new questionnaire based on the results of the first round of interviews and the survey results to get more profound insights into the experimentation processes. For both interview phases, we fostered an exploratory character via a semi-structured interview process. All interviews included the mentioned themes and the discussion of each theme started off with an open question. Except for the first theme, topics were not covered in any particular order, but instead followed the natural flow of the interview.

Selection of participants. We recruited interviewees from industry partners and our own personal networks, and increased our data set using snowball sampling [66], i.e., by asking existing interviewees to put us in contact with further potential interview partners that they are aware of. Key factor for recruiting interview participants was that they have insights into the (technical) details of their company's or project's release process. Therefore, we refrained from interviewing participants in management roles. Another selection criterion was years of experience within the current company. We specified one year of experience as our lower limit for both rounds of interviews.

5.2. Survey

Survey design. To substantiate the findings of the first round of qualitative interviews, we designed an anony-

[7] `https://www.typeform.com/`

mous Web-based survey consisting of, in total, 39 questions. We structured the survey into the three themes release process in general, software deployment (covering the release and experiment evaluation, and roles and responsibilities key concerns), and issues in production (covering quality assurance, and issue handling key concerns). The survey mainly consisted of a combination of multiple-choice, single-choice, and Likert-scale questions. Although the survey had its focus on quantitative aspects, we also included some free-form questions to gain further thoughts and opinions in a more qualitative manner.

Survey participants. In surveys subjects are sampled from a population to which results are intended to be generalized [58]. To address a "tech-savvy" population we distributed the survey within our personal networks (i.e., industry contacts), social media, via two DevOps related newsletters[8,9], and via a German-speaking IT news portal[10]. Survey participants reported an average of 8 years of relevant experience in the software domain (standard deviation 4 years). The resulting participant demographics for the survey is summarized on the bottom part of Figure 3.

5.3. Data Storage

All interviews were audio recorded with the interviewees' approvals. We sent a consent form to the interviewees multiple days prior the interviews containing details on data usage and storage. The audio files and the interview transcriptions generated during the process of data analysis will be stored for 5 years on a university server not accessible by the public. The recorded data will be properly deleted afterwards. The survey data will be exported from the survey platform and kept for five years on the same university server.

6. Analysis

Coding. The first and the second author transcribed the recorded interviews. The first, the second, and the third author coded the transcriptions on a sentence level without a priori codes or categories. For the second round of interviews, we reused codes and added new ones when required. The free-form questions of the survey were coded following the same procedure.

Card sorting. The first three authors analyzed (i.e., investigator triangulation) the qualitative data using open card sorting [67] (683 cards in total), and categorized the participants' statements, resulting in the set of findings presented in Sections 5 and 6. The cards were designed in such a way that each statement was on a single card supplemented with the participant's ID, the actual code, the company type (i.e., startup, SME, corporation), and the application type. The additional information was used to allow for better clustering and identifying differences amongst the various companies or projects involved. All clusters and thus findings are required to be supported by statements of multiple participants.

Chain of evidence. The pre-study was the basis for formulating the interview questions for the first qualitative phase. The card sorting findings of the first qualitative phase formed the basis for the survey questions and response options respectively (e.g., reasons against conducting experiments). We analyzed survey results using the statistical software *R*. The questionnaire of the second round of interviews was based on the analysis of survey results and the findings of the first qualitative phase. All the selected quotes in the paper represent coded statements.

7. Limitations

In Section 4 we present limitations and threats to validity associated with the single phases of our study in detail. An additional limiting factor throughout the interviews is that we only consider data points from a single perspective that are potentially biased having the participants providing idealized data about the CD and experimentation maturity of their companies. In the context of the study it was not possible to analyze additional resources (i.e., data triangulation on a case level) such as process documentation, or deployment scripts. We tried to mitigate this factor by conducting a quantitative online survey to validate the findings of the first qualitative phase.

8. Reporting

Within this paper, we do not only report on the findings of our study, we also provide the reader additional information on the study design (i.e., this case study protocol). Moreover, we provide the interested reader a comprehensive online appendix[11] including all interview materials (i.e., questionnaires), survey questions, survey results in form of a report, and the survey's raw results. We do not expose the names of study participants and the companies they are working for. We used our findings to propose potential directions for future research to the research community.

9. Schedule

The first month of this research was used for planning the study, in the second month we conducted the pre-study. The first round of interviews required two months in total, the transcription of the interviews happened in parallel. Coding and card sorting took another month, similar to the preparation and execution of the survey. We spent a month on writing an initial version of this report. The second round of interviews was conducted in two months. The final data analysis (i.e., coding, card sorting) required us another month. A second version of this report was written afterwards (one month), which was revised three times since then (taking about a month each).

[8] `http://www.devopsweekly.com/`
[9] `http://sreweekly.com/`
[10] `http://heise.de`
[11] `http://www.ifi.uzh.ch/en/seal/people/schermann/projects/expstudy.html`

References

[1] L. Chen, Continuous Delivery: Huge Benefits, but Challenges Too, Software, IEEE 32 (2) (2015) 50–54. doi:10.1109/MS.2015.27.

[2] D. G. Feitelson, E. Frachtenberg, K. L. Beck, Development and Deployment at Facebook, IEEE Internet Computing 17 (4) (2013) 8–17. doi:http://doi.ieeecomputersociety.org/10.1109/MIC.2013.25.

[3] J. Rubin, M. Rinard, The Challenges of Staying Together While Moving Fast: An Exploratory Study, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 982–993. doi:10.1145/2884781.2884871.
URL http://doi.acm.org/10.1145/2884781.2884871

[4] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, Addison-Wesley Professional, 2010.

[5] M. Fowler, Continuous Delivery, http://martinfowler.com/bliki/ContinuousDelivery.html (May 2013).

[6] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, N. Pohlmann, Online Controlled Experiments at Large Scale, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), ACM, New York, NY, USA, 2013, pp. 1168–1176. doi:10.1145/2487575.2488217.
URL http://doi.acm.org/10.1145/2487575.2488217

[7] D. Tang, A. Agarwal, D. O'Brien, M. Meyer, Overlapping experiment infrastructure: More, better, faster experimentation, in: Proceedings 16th Conference on Knowledge Discovery and Data Mining, Washington, DC, 2010, pp. 17–26.

[8] S. Newman, Building Microservices, "O'Reilly Media, Inc.", 2015.

[9] P. Hodgson, Feature Toggles, http://martinfowler.com/articles/feature-toggles.html (Jan. 2016).

[10] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, Y. J. Song, Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, USENIX Association, Berkeley, CA, USA, 2016, pp. 635–650.
URL http://dl.acm.org/citation.cfm?id=3026877.3026926

[11] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, P. Flora, An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 159–168.
URL http://dl.acm.org/citation.cfm?id=2819009.2819034

[12] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, R. Karl, Holistic Configuration Management at Facebook, in: Proceedings of the 25th Symposium on Operating Systems Principles (SOSP), ACM, New York, NY, USA, 2015, pp. 328–343. doi:10.1145/2815400.2815401.
URL http://doi.acm.org/10.1145/2815400.2815401

[13] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.

[14] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, V. Filkov, Quality and Productivity Outcomes Relating to Continuous Integration in GitHub, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 805–816. doi:10.1145/2786805.2786850.
URL http://doi.acm.org/10.1145/2786805.2786850

[15] M. Hilton, T. Tunnell, K. Huang, D. Marinov, D. Dig, Usage, costs, and benefits of continuous integration in open-source projects, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, ACM, New York, NY, USA, 2016, pp. 426–437. doi:10.1145/2970276.2970358.
URL http://doi.acm.org/10.1145/2970276.2970358

[16] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, D. Dig, Trade-offs in continuous integration: Assurance, security, and flexibility, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, ACM, New York, NY, USA, 2017, pp. 197–207. doi:10.1145/3106237.3106270.
URL http://doi.acm.org/10.1145/3106237.3106270

[17] A. Debbiche, M. Dienér, R. Berntsson Svensson, Challenges when adopting continuous integration: A case study, in: Product-Focused Software Process Improvement: 15th International Conference, PROFES 2014, Helsinki, Finland., Springer International Publishing, 2014, pp. 17–32.

[18] M. Brandtner, E. Giger, H. Gall, Supporting continuous integration by mashing-up software quality information, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 184–193. doi:10.1109/CSMR-WCRE.2014.6747169.

[19] D. Ståhl, J. Bosch, Modeling continuous integration practice differences in industry software development, Journal of Systems and Software 87 (2014) 48 – 59. doi:http://dx.doi.org/10.1016/j.jss.2013.08.032.
URL http://www.sciencedirect.com/science/article/pii/S0164121213002276

[20] M. Beller, G. Gousios, A. Zaidman, Oops, my tests broke the build: An explorative analysis of travis ci with github, in: Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 356–367. doi:10.1109/MSR.2017.62.
URL https://doi.org/10.1109/MSR.2017.62

[21] T. Rausch, W. Hummer, P. Leitner, S. Schulte, An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software, in: 14th International Conference on Mining Software Repositories (MSR'17), 2017.

[22] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. D. Penta, S. Panichella, A tale of ci build failures: an open source and a financial organization perspective, in: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, p. nn.

[23] B. Adams, S. McIntosh, Modern Release Engineering in a Nutshell – Why Researchers should Care, in: Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), Future of Software Engineering (FOSE) Track, 2016.

[24] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, M. Oivo, Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study, Journal of Systems and Software-doi:http://dx.doi.org/10.1016/j.jss.2015.12.015.
URL http://www.sciencedirect.com/science/article/pii/S0164121215002812

[25] M. Shahin, M. A. Babar, L. Zhu, Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, IEEE Access 5 (2017) 3909–3943. doi:10.1109/ACCESS.2017.2685629.

[26] A. Rahman, E. Helms, L. Williams, C. Parnin, Synthesizing Continuous Deployment Practices Used in Software Development, in: Agile Conference (AGILE), 2015, 2015, pp. 1–10. doi:10.1109/Agile.2015.12.

[27] ThoughtWorks and Forrester Consulting, Continuous Delivery: A Maturity Assessment Model, https://info.thoughtworks.com/Continuous-Delivery-Maturity-Model.html (2013).

[28] Puppet Labs, State of DevOps Report, https://puppetlabs.com/2016-devops-report (2016).

[29] J. Cito, P. Leitner, T. Fritz, H. C. Gall, The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, 2015, pp. 393–403. doi:10.1145/2786805.2786826.
URL http://doi.acm.org/10.1145/2786805.2786826

[30] D. Bruneo, T. Fritz, S. Keidar-Barner, P. Leitner, F. Longo, C. Marquezan, A. Metzger, K. Pohl, A. Puliafito, D. Raz, A. Roth, E. Salant, I. Segall, M. Villari, Y. Wolfsthal, C. Woods, CloudWave: where Adaptive Cloud Management Meets DevOps, in: Proceedings of the Fourth International Workshop on Management of Cloud Systems (MoCS 2014), 2014.

[31] L. E. Lwakatare, P. Kuvaja, M. Oivo, Dimensions of devops, in: International Conference on Agile Software Development, Springer, 2015, pp. 212–217.

[32] L. E. Lwakatare, P. Kuvaja, M. Oivo, Relationship of devops to agile, lean and continuous deployment: A multivocal literature review study, in: Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway., Springer, 2016, pp. 399–415.

[33] M. Shahin, M. Zahedi, M. A. Babar, L. Zhu, Adopting continuous delivery and deployment: Impacts on team structures, collaboration and responsibilities, in: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, ACM, 2017, pp. 384–393.

[34] M. Leppanen, S. Makinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. Mantyla, T. Mannisto, The Highways and Country Roads to Continuous Deployment, IEEE Software 32 (2) (2015) 64–72. doi:10.1109/MS.2015.50.

[35] H. Olsson, H. Alahyari, J. Bosch, Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software, in: Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), 2012, pp. 392–399. doi:10.1109/SEAA.2012.54.

[36] S. Neely, S. Stolt, Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy), in: Agile Conference (AGILE), 2013, 2013, pp. 121–128. doi:10.1109/AGILE.2013.17.

[37] G. G. Claps, R. B. Svensson, A. Aurum, On the Journey to Continuous Deployment: Technical and Social Challenges Along the Way, Information and Software Technology 57 (0) (2015) 21 – 31. doi:http://dx.doi.org/10.1016/j.infsof.2014.07.009.

[38] L. Chen, Continuous delivery: Overcoming adoption challenges, Journal of Systems and Software (2017) – doi:http://dx.doi.org/10.1016/j.jss.2017.02.013.
URL http://www.sciencedirect.com/science/article/pii/S0164121217300353

[39] S. Bellomo, N. Ernst, R. Nord, R. Kazman, Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail, in: Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2014, pp. 702–707. doi:10.1109/DSN.2014.104.

[40] J. Itkonen, R. Udd, C. Lassenius, T. Lehtonen, Perceived benefits of adopting continuous delivery practices, in: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '16, ACM, New York, NY, USA, 2016, pp. 42:1–42:6. doi:10.1145/2961111.2962627.
URL http://doi.acm.org/10.1145/2961111.2962627

[41] B. Fitzgerald, K.-J. Stol, Continuous software engineering and beyond: Trends and challenges, in: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, RCoSE 2014, ACM, New York, NY, USA, 2014, pp. 1–9. doi:10.1145/2593812.2593813.
URL http://doi.acm.org/10.1145/2593812.2593813

[42] G. Schermann, J. Cito, P. Leitner, H. C. Gall, Towards Quality Gates in Continuous Delivery and Deployment, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), IEEE, 2016, pp. 1–4.

[43] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, M. Stumm, Continuous Deployment at Facebook and OANDA, in: Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16, ACM, New York, NY, USA, 2016, pp. 21–30. doi:10.1145/2889160.2889223.
URL http://doi.acm.org/10.1145/2889160.2889223

[44] K. Kevic, B. Murphy, L. Williams, J. Beckmann, Characterizing experimentation in continuous deployment: a case study on bing, in: International Conference on Software Engineering, Software Engineering in Practice, ICSE SEIP, Buenos Aires, 2017.

[45] A. Fabijan, P. Dmitriev, H. H. Olsson, J. Bosch, The evolution of continuous experimentation in software product development, in: International Conference on Software Engineering, ICSE, Buenos Aires, 2017.

[46] F. Fagerholm, A. S. Guinea, H. Mäenpää, J. Münch, The right model for continuous experimentation, Journal of Systems and Software 123 (2017) 292 – 305. doi:http://dx.doi.org/10.1016/j.jss.2016.03.034.
URL http://www.sciencedirect.com/science/article/pii/S0164121216300024

[47] E. Bakshy, E. Frachtenberg, Design and Analysis of Benchmarking Experiments for Distributed Internet Services, in: Proceedings of the 24th International Conference on World Wide Web (WWW), 2015, pp. 108–118. doi:10.1145/2736277.2741082.
URL http://doi.acm.org/10.1145/2736277.2741082

[48] E. Bakshy, D. Eckles, M. S. Bernstein, Designing and Deploying Online Field Experiments, in: Proceedings of the 23rd International Conference on World Wide Web, WWW '14, ACM, New York, NY, USA, 2014, pp. 283–292. doi:10.1145/2566486.2567967.
URL http://doi.acm.org/10.1145/2566486.2567967

[49] R. Kohavi, R. Longbotham, D. Sommerfield, R. M. Henne, Controlled experiments on the web: survey and practical guide, Data Mining and Knowledge Discovery 18 (1) (2009) 140–181. doi:10.1007/s10618-008-0114-1.
URL http://dx.doi.org/10.1007/s10618-008-0114-1

[50] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, I. Baldini, CanaryAdvisor: A Statistical-based Tool for Canary Testing (Demo), in: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA), ACM, New York, NY, USA, 2015, pp. 418–422. doi:10.1145/2771783.2784770.
URL http://doi.acm.org/10.1145/2771783.2784770

[51] G. Tamburrelli, A. Margara, Towards Automated A/B Testing, in: Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE), Vol. 8636 of Lecture Notes in Computer Science, Springer, 2014, pp. 184–198.

[52] M. T. Rahman, L.-P. Querel, P. C. Rigby, B. Adams, Feature Toggles: Practitioner Practices and a Case Study, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, ACM, New York, NY, USA, 2016, pp. 201–211. doi:10.1145/2901739.2901745.
URL http://doi.acm.org/10.1145/2901739.2901745

[53] G. Schermann, D. Schöni, P. Leitner, H. C. Gall, Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies, in: Proceedings of the 17th International Middleware Conference, Middleware '16, ACM, New York, NY, USA, 2016, pp. 12:1–12:14. doi:10.1145/2988336.2988348.
URL http://doi.acm.org/10.1145/2988336.2988348

[54] E. Lindgren, J. Münch, Raising the Odds of Success: the Current State of Experimentation in Product Development, Information and Software Technology 77 (2016) 80 – 91. doi:http://dx.doi.org/10.1016/j.infsof.2016.04.008.
URL http://www.sciencedirect.com/science/article/pii/S0950584916300647

[55] M. Shahin, M. A. Babar, L. Zhu, The intersection of continuous deployment and architecting process: Practitioners' perspectives, in: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '16, ACM, New York, NY, USA, 2016, pp. 44:1–44:10. doi:10.1145/2961111.2962587.
URL http://doi.acm.org/10.1145/2961111.2962587

[56] F. Shull, J. Singer, D. I. Sjøberg, Guide to Advanced Empirical Software Engineering, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[57] P. Brereton, B. Kitchenham, D. Budgen, Z. Li, Using a protocol template for case study planning, in: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, EASE'08, BCS Learning & Development Ltd., Swindon, UK, 2008, pp. 41–48.
URL http://dl.acm.org/citation.cfm?id=2227115.2227120

[58] P. Runeson, M. Höst, A. Rainer, B. Regnell, Case Study Research in Software Engineering: Guidelines and Examples, 1st Edition, Wiley Publishing, 2012.

[59] V. Garousi, M. Felderer, M. V. Mäntylä, The need for multivocal literature reviews in software engineering: Complementing systematic literature reviews with grey literature, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE '16, ACM, New York, NY, USA, 2016, pp. 26:1–26:6. doi:10.1145/2915970.2916008.
URL http://doi.acm.org/10.1145/2915970.2916008

[60] Facebook, Engineering Blog, https://code.facebook.com/posts/ (2016).

[61] Etsy, Code as Craft, https://codeascraft.com/ (2016).

[62] Twitter, The Twitter Engineering Blog, https://blog.twitter.com/engineering (2016).

[63] Google, Google Developers Blog, https://developers.googleblog.com/ (2016).

[64] Netflix, The Netflix Tech Blog, http://techblog.netflix.com/ (2016).

[65] T. Barik, B. Johnson, E. Murphy-Hill, I Heart Hacker News: Expanding Qualitative Research Findings by Analyzing Social News Websites, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, 2015, pp. 882–885. doi:10.1145/2786805.2803200.
URL http://doi.acm.org/10.1145/2786805.2803200

[66] R. Atkinson, J. Flint, Accessing hidden and hard-to-reach populations: Snowball research strategies, Social research update 33 (1) (2001) 1–4.

[67] D. Spencer, Card Sorting: Designing Usable Categories, Rosenfeld Media, 2009.

[68] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, T. Zimmermann, Improving Developer Participation Rates in Surveys, in: Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), 2013, pp. 89–92. doi:10.1109/CHASE.2013.6614738.

[69] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, 2000.

[70] G. Schermann, J. Cito, P. Leitner, All the Services Large and Micro: Revisiting Industrial Practice in Services Computing, in: Proceedings of the 11th International Workshop on Engineering Service Oriented Applications (WESOA'15), 2015.

[71] G. Mazlami, J. Cito, P. Leitner, Extraction of microservices from monolithic software architectures, in: 2017 IEEE International Conference on Web Services (ICWS), 2017, pp. 524–531. doi:10.1109/ICWS.2017.61.

[72] P. Kruchten, R. L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, IEEE Software 29 (6) (2012) 18–21. doi:10.1109/MS.2012.167.

[73] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, W. Shang, Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, ACM, New York, NY, USA, 2016, pp. 1–12. doi:10.1145/2901739.2901774.
URL http://doi.acm.org/10.1145/2901739.2901774

[74] B. Lin, A. Zagalsky, M. Storey, A. Serebrenik, Why Developers Are Slacking Off: Understanding How Software Teams Use Slack, in: Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW) Companion, ACM, New York, NY, USA, 2016, pp. 333–336. doi:10.1145/2818052.2869117.
URL http://doi.acm.org/10.1145/2818052.2869117

[75] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, A. Roth, Runtime Metric Meets Developer - Building Better Cloud Applications Using Feedback, in: Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015), ACM, New York, NY, USA, 2015.

[76] A. Moskowitz, Eat Your Own Dog Food, j-LOGIN 28 (5).
URL http://www.usenix.org/publications/login/2003-10/pdfs/moskowitz.pdf

[77] M. Galster, D. Weyns, D. Tofan, B. Michalik, P. Avgeriou, Variability in software systems - a systematic literature review, IEEE Transactions on Software Engineering 40 (3) (2014) 282–306. doi:10.1109/TSE.2013.56.

[78] R. Capilla, J. Bosch, K.-C. Kang, et al., Systems and software variability management, Concepts Tools and Experiences.

[79] G. Kiczales, Aspect-oriented Programming, ACM Computing Surveys 28 (4). doi:10.1145/242224.242420.
URL http://doi.acm.org/10.1145/242224.242420

[80] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, Dynamic Software Product Lines, Computer 41 (4) (2008) 93–95. doi:10.1109/MC.2008.123.

[81] M. Kim, T. Zimmermann, R. DeLine, A. Begel, The Emerging Role of Data Scientists on Software Development Teams, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 96–107. doi:10.1145/2884781.2884783.
URL http://doi.acm.org/10.1145/2884781.2884783

[82] G. Hohpe, I. Ozkaya, U. Zdun, O. Zimmermann, The software architect's role in the digital age, IEEE Software 33 (6) (2016) 30–39. doi:10.1109/MS.2016.137.

[83] W. Hummer, F. Rosenberg, F. Oliveira, T. Eilam, Testing Idempotence for Infrastructure as Code, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 368–388. doi:10.1007/978-3-642-45065-5_19.
URL http://dx.doi.org/10.1007/978-3-642-45065-5_19

[84] K. Braa, R. Vidgen, Interpretation, intervention, and reduction in the organizational laboratory: a framework for in-context information system research, Accounting, Management and Information Technologies 9 (1) (1999) 25 – 47. doi:https://doi.org/10.1016/S0959-8022(98)00018-6.
URL http://www.sciencedirect.com/science/article/pii/S0959802298000186

[85] T. C. Lethbridge, S. E. Sim, J. Singer, Studying software engineers: Data collection techniques for software field studies, Empirical Software Engineering 10 (3) (2005) 311–341. doi:10.1007/s10664-005-1290-x.
URL https://doi.org/10.1007/s10664-005-1290-x